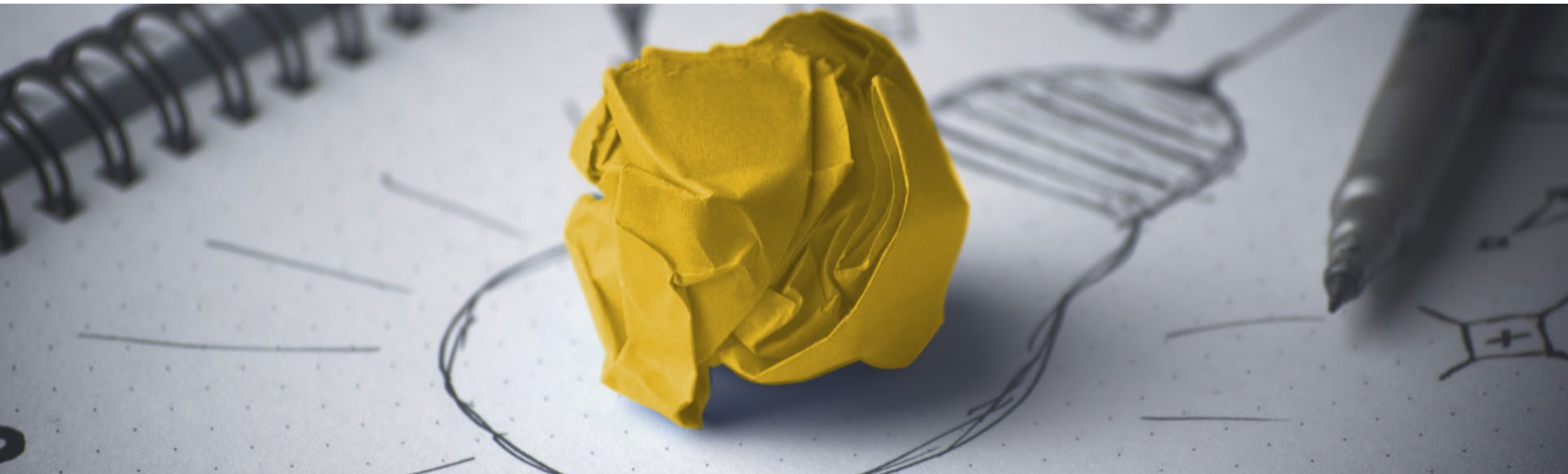


# Grundlagen

Engineering & TypeScript



- Vorstellungen
- TypeScript
  - Allgemein
  - Modules, Interfaces, Classes
  - Generics, Decorators
- Engineering
  - Design Patterns
- Tooling
  - NPM, Bower, Typings



Die 2BIT GmbH vereinfacht die technologische Nutzung der Enduser in der Sprache des Kunden auf allen Geräten.

# Vorstellung Raphael Ritter

---

- Ausbildung
  - HTW 2013 – 2014 EMBA
  - HSR 2007 – 2010 BSc. Computer Science
  - Informatikmittelschule Chur 2002 - 2006
- Managing Partner 2BIT GmbH
  - Web und App Entwicklung (Progressive Web Apps / NativeScript)
  - Business Intelligence / Data Science
  - Consulting
  - Teaching HSR / HSLU

# Vorstellung Felix Egli

---

- Ausbildung
  - HSR 2008 – 2011 BSc. Computer Science
  - Applikationsentwickler Lehre 2001 - 2005
- Jobs
  - 2019 – Dozent 2bit Angular Kurse
  - 2014 – Heute CAS Dozent an der HSR
  - 2014 – Heute      Geschäftsführer cross performance GmbH
  - 2011 – 2015      Zühlke Engineering AG
  - 2005 – 2007      Crealogix AG

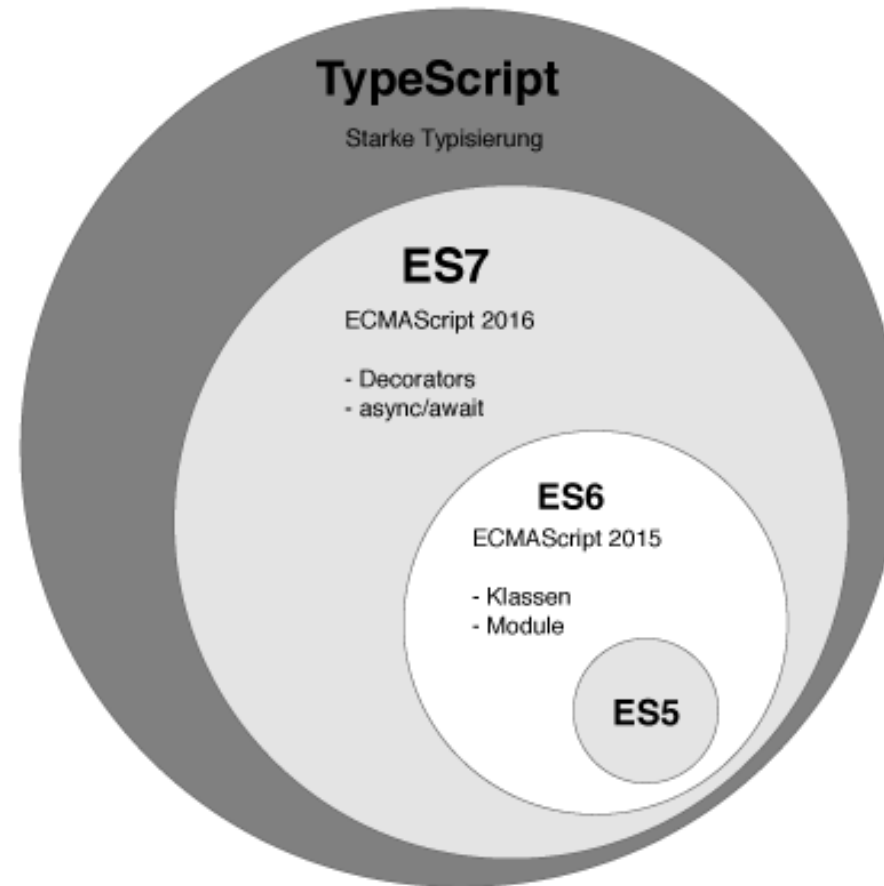
# TypeScript – Fragen

---

- Was ist der Vorteil von TypeScript?
- Warum sollte ich TypeScript verwenden oder nicht?
- Was ist der grösste Unterschied zu normalem JavaScript?

# TypeScript – Kontext

---



# TypeScript – Kontext

---

- TypeScript ist ein Super Set von JavaScript und wird schlussendlich zu normalen JavaScript kompiliert
- TypeScript fügt der Web Entwicklung Typensicherheit hinzu, welches gerade grösseren Teams eine grosse Hilfe ist
- TypeScript ermöglicht es Entwicklern bereits heute Features von zukünftigen Sprachdefinitionen wie ES7 zu verwenden
  - Async und decorators sind nur einige Beispiele davon

# TypeScript - Geschichte

---

- Die erste Version von TypeScript ist im Jahre 2012 (Version 0.8) erschienen
  - Die momentan aktuelle Version ist 3.9
- TypeScript wird immer populärer dank seiner Vorteile
  - Angular ist komplett in TypeScript geschrieben und nutzt dessen Vorteile konsequent aus
- Die Features von TypeScript und deren Sprach Konstrukte findet ihr hier: <https://www.typescriptlang.org/docs/>



# TypeScript – Basis Typen

- Es gibt folgende Basis-Typen
  - boolean, number, string, Array, Tuple, Enum, void, any

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ['hello', 10]; // OK
// Initialize it incorrectly
x = [10, 'hello']; // Error
```

```
enum Color {Red, Green, Blue};
let c: Color = Color.Green;
```

```
let name: string = `Gene`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ name }.
I'll be ${ age + 1 } years old next month.`
```

```
let list: Array<number> = [1, 2, 3];
```

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

# TypeScript – Enums

- Mit enums können wie in Hochsprachen numerische Konstanten zusammengefasst werden

```
enum Direction {  
    Up = 1,  
    Down,  
    Left,  
    Right  
}
```

```
enum Enum {  
    A  
}  
let a = Enum.A;  
let nameOfA = Enum[Enum.A]; // "A"
```



```
var Enum;  
(function (Enum) {  
    Enum[Enum["A"] = 0] = "A";  
})(Enum || (Enum = {}));  
var a = Enum.A;  
var nameOfA = Enum[Enum.A]; // "A"
```

# TypeScript – Array

---

- Wie sieht eine Definition von einem Array aus mit Zahlen gefüllt
  - Direkt initialisiert mit 2,3,4?
- Playground do it
  - <http://www.typescriptlang.org/Playground/>
- `let numbers = new Array<number>(1,2,3);`
- `let numbers1 = [1,2,3];`



# TypeScript – Casting

- Es gibt zwei Möglichkeiten in TypeScript etwas zu casten
  - <> Syntax `let strLength: number = (<string>someValue).length;`
  - as Syntax `let strLength: number = (someValue as string).length;`
- Es ist grundsätzlich der «as» Syntax zu bevorzugen, da nur dieser Syntax auch in jsx files funktioniert


# TypeScript – Type Guards

- Type Guards können verwendet werden um dem Compiler zusätzliche Typeninformationen zu liefern um Casting zu verhindern

```
interface Bird {  
    fly();  
    layEggs();  
}  
  
interface Fish {  
    swim();  
    layEggs();  
}  
  
function getSmallPet(): Fish | Bird {  
    // ...  
}
```

```
function isFish(pet: Fish | Bird): pet is Fish {  
    return (<Fish>pet).swim !== undefined;  
}
```

```
let pet = getSmallPet();  
  
if ((<Fish>pet).swim) {  
    (<Fish>pet).swim();  
}  
else {  
    (<Bird>pet).fly();  
}
```



```
if (isFish(pet)) {  
    pet.swim();  
}  
else {  
    pet.fly();  
}
```

# TypeScript – Klassen

---

- Klassen, Interfaces, Abstrakte Klassen
  - Funktionen innerhalb der Klassen können die Sichtbarkeit public, protected und private haben
  - Properties können static sein
  - Properties innerhalb von Klassen können (müssen aber nicht) mit getter und setter der Aussenwelt zur Verfügung gestellt werden
  - Interfaces werden von Klassen implementiert
  - Klassen können einander vererbt

# TypeScript/ES6 – Klassen

```
class Animal {  
    private name: string;  
    constructor(theName: string) { this.name = theName; }  
}  
  
class Rhino extends Animal {  
    constructor() { super("Rhino"); }  
}
```

```
class Animal {  
    constructor(private name: string) { }  
    move(distanceInMeters: number) {  
        console.log(`${this.name} moved ${distanceInMeters}m.`);  
    }  
}
```

```
class Person {  
    protected name: string;  
    constructor(name: string) { this.name = name; }  
}  
  
class Employee extends Person {  
    private department: string;  
  
    constructor(name: string, department: string) {  
        super(name);  
        this.department = department;  
    }  
  
    public getElevatorPitch() {  
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;  
    }  
}
```

# TypeScript/ES6 – Klassen

```
class Employee {
  private _fullName: string;

  get fullName(): string {
    return this._fullName;
  }

  set fullName(newName: string) {
    if (passcode && passcode == "secret passcode") {
      this._fullName = newName;
    }
    else {
      console.log("Error: Unauthorized update of employee!");
    }
  }
}
```

```
abstract class Animal {
  abstract makeSound(): void;
  move(): void {
    console.log('roaming the earth...');
  }
}

class Grid {
  static origin = {x: 0, y: 0};
  calculateDistanceFromOrigin(point: {x: number; y: number;}) {
    let xDist = (point.x - Grid.origin.x);
    let yDist = (point.y - Grid.origin.y);
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
  }
  constructor (public scale: number) { }
}

let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
```



# TypeScript – Optional Chaining

- Optional Chaining für einfache Null / undefined checks

```
let x = (foo === null || foo === undefined) ?  
    undefined :  
    foo.bar.baz();
```



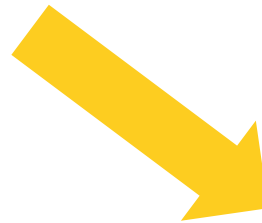
```
let x = foo?.bar.baz();
```

```
// Before  
if (foo && foo.bar && foo.bar.baz) {  
    // ...  
}  
  
// After-ish  
if (foo?.bar?.baz) {  
    // ...  
}
```

# TypeScript – Null Coalescing

- Falls eine Variable null / undefined ist soll ein Default Wert zurück gegeben werden

```
let x = (foo !== null && foo !== undefined) ?  
    foo :  
    bar();
```



```
let x = foo ?? bar();
```

# TypeScript - Übung

---

- Schaut euch auf dem Playground folgende Beispiele an und verändert diese
  - JavaScript Essentials
  - Functions with JavaScript
  - Working with Classes
  - Modern JavaScript

<http://www.typescriptlang.org/play/index.html>



# TypeScript - Module

---

- Module
  - Module werden nicht im global Scope ausgeführt sondern in ihrem eigenen Scope
  - Falls ein Modul Funktionalität eines anderen konsumieren möchte, muss dieses zuerst importiert werden
  - Mit dem default Keyword kann die Funktion angegeben werden, welche als default bei einem Import importiert wird

<https://www.typescriptlang.org/docs/handbook/modules.html>

# TypeScript – Module

```
class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
        return s.length === 5 && numberRegexp.test(s);  
    }  
}  
  
export { ZipCodeValidator };  
export { ZipCodeValidator as mainValidator };
```

```
import { ZipCodeValidator as ZCV } from "../ZipCodeValidator";  
let myValidator = new ZCV();
```

```
import * as validator from "../ZipCodeValidator";  
let myValidator = new validator.ZipCodeValidator();
```

```
declare let $: JQuery;  
export default $;
```

```
import $ from "jQuery";
```

```
$("button.continue").html( "Next Step..." );
```

# TypeScript – Module

- Es gibt verschiedene Möglichkeiten welcher Modulloader verwendet werden soll
  - CommonJS
  - SystemJS
  - Webpack
  - ECMA2015
  - ...
- Die konkrete Umsetzung davon macht der TypeScript Compiler, wodurch beliebig zwischen den verschiedenen Frameworks gewechselt werden kann

```
"compilerOptions": {  
  "module": "commonjs",  
  "noImplicitAny": true,  
  "removeComments": true,  
  "preserveConstEnums": true,  
  "outFile": "../../built/local/tsc.js",  
  "sourceMap": true  
},
```

# TypeScript – Moduleloader Beispiele

- Je nach Compiler Einstellungen

*SimpleModule.ts*

```
import m = require("mod");  
export let t = m.something + 1;
```

*AMD / RequireJS SimpleModule.js*

```
define(["require", "exports", "./mod"], function (require, exports, mod_1) {  
    exports.t = mod_1.something + 1;  
});
```

*CommonJS / Node SimpleModule.js*

```
var mod_1 = require("./mod");  
exports.t = mod_1.something + 1;
```

# TypeScript – Generics

---

- Mit Generics können Typen geschaffen werden, welche mit unterschiedlichen Klassen arbeiten können
- Das beste Beispiel für den Einsatz von Generics sind Listen, da diese unabhängig von ihrem Inhalt funktionieren

```
function identity<T>(arg: T): T {  
    return arg;  
}
```

```
let output = identity<string>("myString"); // type of output will be 'string'
```



# TypeScript – Übungen Generics

- Schaut euch auf dem Playground folgendes Beispiel an und verändert dieses
  - Generic Classes
    - Versuche einen neuen Typ für den Kleiderschrank zu erstellen
    - Füge eine neue Instanz dieses Types dem Kleiderschrank hinzu

<http://www.typescriptlang.org/Playground/>



# TypeScript/ES6 – Arrow Function

- Mit TypeScript können auch Lambda Expressions (Arrow Functions genannt in TypeScript) verwendet werden
- Dadurch können Funktionen sehr einfach in einer Kurzschreibweise deklariert werden

```
let deck = {  
  suits: ["hearts", "spades", "clubs", "diamonds"],  
  cards: Array(52),  
  createCardPicker: function() {  
    // Notice: the line below is now a lambda, allowing us to capture `this` earlier  
  
    return () => {  
      let pickedCard = Math.floor(Math.random() * 52);  
      let pickedSuit = Math.floor(pickedCard / 13);  
  
      return {suit: this.suits[pickedSuit], card: pickedCard % 13};  
    }  
  }  
}
```

# TypeScript – TS Config

- tsconfig.json
  - Compiler Konfiguration
  - Projektverzeichnis
- Anwenden
  - tsc (sucht nach tsconfig.json)
  - tsc --project *DIR*

```

{} tsconfig.json x {} tsconfig.app.json
1  {
2    "compileOnSave": false,
3    "compilerOptions": {
4      "baseUrl": "./",
5      "importHelpers": true,
6      "outDir": "./dist/out-tsc",
7      "sourceMap": true,
8      "declaration": false,
9      "moduleResolution": "node",
10     "emitDecoratorMetadata": true,
11     "experimentalDecorators": true,
12     "target": "es5",
13     "typeRoots": [
14       "node_modules/@types"
15     ],
16     "lib": [
17       "es2017",
18       "dom"
19     ]
20   }
21 }

```

Erweiterung:

```

{} tsconfig.json x {} tsconfig.app.json x
1  {
2    "extends": "../..tsconfig.json",
3    "compilerOptions": {
4      "outDir": "../..out-tsc/app",
5      "module": "es2015",
6      "types": []
7    },
8    "exclude": [
9      "src/test.ts",
10     "**/*.spec.ts"
11   ]
12 }

```

# TypeScript – Decorators

---

- Decorators werden verwendet um Klassen, Methoden oder Properties dynamisch zu erweitern
- Es gibt vier Arten von Decorators, welche in TypeScript verwendet werden können
  - Class, Method, Property, Parameter
- Ein Decorator ist schlussendlich ein Funktionsaufruf in JavaScript welcher anstelle der ursprünglichen Funktion aufgerufen wird
- Experimental! -> Muss speziell konfiguriert werden (tsconfig.json)

# TypeScript – Decorators

@myDecorator

Function / Class

Decorated

**Decorator**

precondition

Altered fnc

Function / Class

postcondition

# TypeScript – Class Decorator

- Der Class Decorator wird auf die Klassendeklaration angewendet und greift auf den Konstruktor
- Mit dem Rückgabewert kann der Konstruktor ersetzt werden um das Verhalten anzupassen

```
function sealed(constructor: Function) {  
    Object.seal(constructor);  
    Object.seal(constructor.prototype);  
}
```

```
declare type ClassDecorator = <TFunction extends Function>  
(target: TFunction) => TFunction | void;
```

```
@sealed  
class Greeter {  
    greeting: string;  
    constructor(message: string) {  
        this.greeting = message;  
    }  
    greet() {  
        return "Hello, " + this.greeting;  
    }  
}
```

# TypeScript – Method Decorator

- Der Method Decorator wird dem Methodenaufruf angefügt
- Mit ihm kann die ursprüngliche Funktion mit einer anderen Funktion ausgetauscht (im Normalfall erweitert) werden

```
declare type MethodDecorator = <T>(target: Object, propertyKey:  
string | symbol, descriptor: TypedPropertyDescriptor<T>) =>  
TypedPropertyDescriptor<T> | void;
```

```
function enumerable(value: boolean) {  
    return function (target: any, propertyKey: string, descriptor: PropertyDescript  
or) {  
        descriptor.enumerable = value;  
    };  
}
```

```
class Greeter {  
    greeting: string;  
    constructor(message: string) {  
        this.greeting = message;  
    }  
  
    @enumerable(false)  
    greet() {  
        return "Hello, " + this.greeting;  
    }  
}
```

# TypeScript – Property Decorator

- Property Decorators werden auf Properties angewendet

```
class Greeter {  
    @format("Hello, %s")  
    greeting: string;  
  
    constructor(message: string) {  
        this.greeting = message;  
    }  
    greet() {  
        let formatString = getFormat(this, "greeting");  
        return formatString.replace("%s", this.greeting);  
    }  
}
```

```
declare type PropertyDecorator = (target: Object, propertyKey:  
string | symbol) => void;
```

```
import "reflect-metadata";  
  
const formatMetadataKey = Symbol("format");  
  
function format(formatString: string) {  
    return Reflect.metadata(formatMetadataKey, formatString);  
}  
  
function getFormat(target: any, propertyKey: string) {  
    return Reflect.getMetadata(formatMetadataKey, target, propertyKey);  
}
```



# TypeScript – Parameter Decorator

- Parameter Decorators können direkt auf Funktionsparameter angewandt werden

```
import "reflect-metadata";

const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol, parameterIndex: number) {
    let existingRequiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey) || [];
    existingRequiredParameters.push(parameterIndex);
    Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters, target, propertyKey);
}

declare type ParameterDecorator = (target: Object, propertyKey: string | symbol, parameterIndex: number) => void;
```

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    @validate
    greet(@required name: string) {
        return "Hello " + name + ", " + this.greeting;
    }
}
```

# TypeScript – Übungen Decorators

- Schaut euch die beiden Übungen zum Class und Method Decorator an. Gemeinsam versuchen wir die vorhandenen Todo's in der Ausgangslage auszuprogrammieren

<https://github.com/rdnscr/2school-typescript>



# TypeScript – Async / Await

- Asynchrone Programmierung ist kompliziert und der Programmablauf ist schwierig nachzuverfolgen

```
readFromBackendOne.then((result) => {
    console.log(result);
    readFromBackendAfterOne.then((result) => {
        console.log(result);
        readFromBackendAfterAfterOne.then((result) => {
            console.log(result);
        });
    });
});
```

VS

```
async function asyncExample() {
    console.log(await readFromBackendOne);
    console.log(await readFromBackendAfterOne);
    console.log(await readFromBackendAfterAfterOne);
}

asyncExample();
```

# TypeScript – Async / Await

- Promise ONLY
  - Asynchron => Synchron
  - Nur Syntactic Sugar

```
getPersonFullNameUsingThen() {  
    return fetch('./data/person.json')  
        .then((response: any) => {  
            return response.json();  
        })  
        .then((person: any) => {  
            console.log(`${person.firstName} ${person.lastName}`);  
        })  
}
```

```
async getPersonFullNameUsingAsync() {  
    let response = await fetch('./data/person.json');  
    let person = await response.json();  
    console.log(`${person.firstName} ${person.lastName}`);  
}
```

# TypeScript – Async / Await

- Output?

```
function delay(milliseconds: number) {  
    return new Promise<void>(resolve => {  
        setTimeout(resolve, milliseconds);  
    });  
}  
  
async function dramaticWelcome() {  
    console.log("Hello");  
  
    for (let i = 0; i < 3; i++) {  
        await delay(500);  
        console.log(".");  
    }  
  
    console.log("World!");  
}  
  
dramaticWelcome();
```

# TypeScript – Async / Await - Practice

- <https://www.typescriptlang.org/docs/handbook/release-notes/typescript-1-7.html>
- Baue das printDelayed selbst nach

```
function delay(milliseconds: number) {  
    return new Promise<void>(resolve => {  
        setTimeout(resolve, milliseconds);  
    });  
}  
  
async function dramaticWelcome() {  
    console.log("Hello");  
  
    for (let i = 0; i < 3; i++) {  
        await delay(500);  
        console.log(".");  
    }  
  
    console.log("World!");  
}  
  
dramaticWelcome();
```



# TypeScript – Definition Files

---

- Wir wollen Frameworks verwenden ohne die Typensicherheit zu verlieren
- Dafür gibt es in TypeScript die Möglichkeit Definition Files zu erstellen
- Die Definition Files beschreiben die API einer Bibliothek
  - Ein Definition File besteht meistens aus verschiedenen Namespaces (Modulen) und den entsprechenden Interfaces über die kommuniziert wird

# TypeScript – Definition Files

---

- Es gibt online ein GitHub Repository mit einer Ansammlung von Deklarationsfiles
  - In dieser Sammlung findet man zu jeder halbwegs populären Bibliothek ein Definition File
  - Solltet ihr für eure eingesetzten JavaScript Frameworks / Bibliotheken kein Definition File finden, würde ich mir der Einsatz der Bibliothek nochmals überlegen ;-)

<https://github.com/DefinitelyTyped/DefinitelyTyped>

<http://www.typescriptlang.org/Handbook#writing-dts-files>



- JavaScript in TypeScript umwandeln



- Übung.js => Lösung.ts
- <https://github.com/rdnscr/2school-typescript>

# Engineering

---

- DRY (Don't Repeat Yourself)
- YAGNI (You Ain't Gonna Need It)
- KISS (Keep It Simple and Stupid)

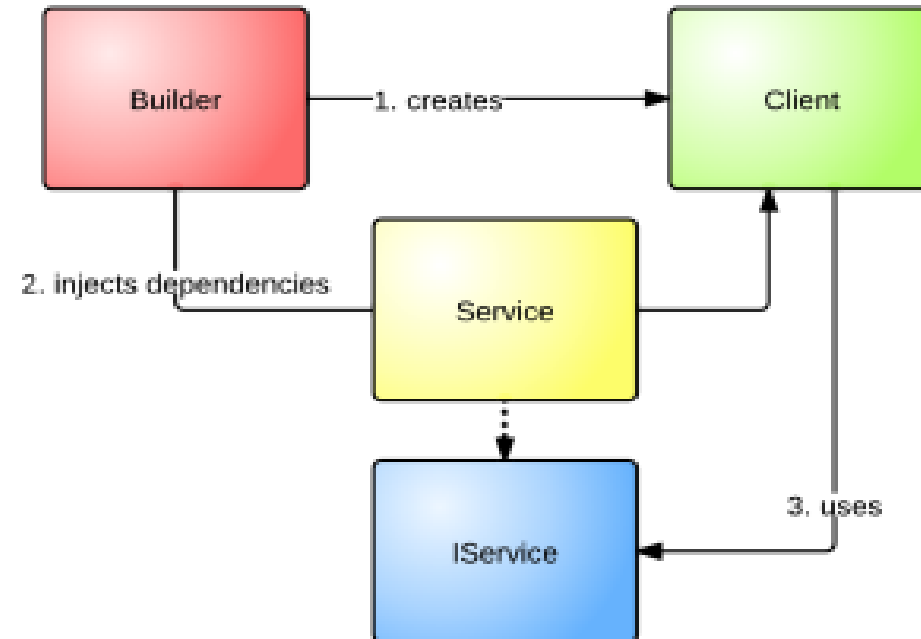
- Schaut euch das TypeScript File Principals\_Ausgangslage.ts an
- Welches Prinzip wird hier verletzt?
- Programmiert in TypeScript eine saubere Lösung davon



- SOLID
  - Single responsibility principle
    - Klasse / Funktion dient einem gezieltem Zweck
  - Open/Close principle (Open for extension, closed for modification)
    - Das Verhalten kann erweitert aber nicht modifiziert werden
  - Liskov substitution principle
    - Es werden klare «Verträge» ausgehandelt die eingehalten werden müssen egal welche Implementation verwendet wird (Design by Contract)
  - Interface segregation principle
    - Lieber mehrere spezifische Interfaces als ein riesiges generelles
  - Dependency inversion principle

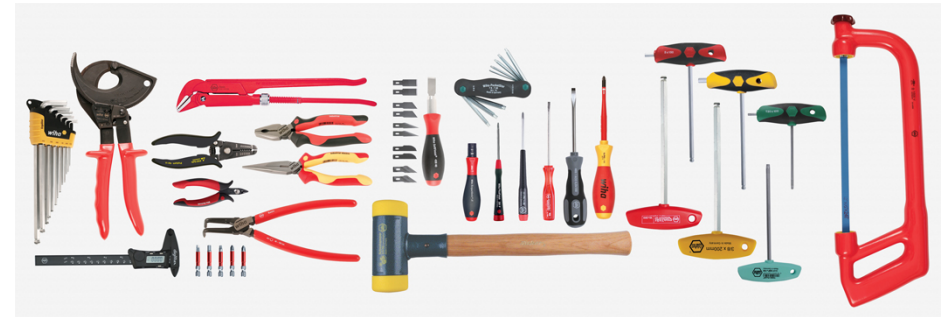
- Schaut euch die Übung 1 zu SOLID an. Um welches Prinzip geht es hier?
  - Programmiert die korrekte Lösung in TypeScript
- Schaut euch die Übung 2 zu SOLID an. Um welches Prinzip geht es hier?
  - Programmiert die korrekte Lösung in TypeScript
- Schaut euch die Übung 3 zu SOLID an. Um welches Prinzip geht es hier?
  - Programmiert die korrekte Lösung in TypeScript

- Dependency Injection
  - Reduzieren von hoher Kopplung zwischen verschiedenen Klassen





Alles wird im Container registriert,  
damit es jeder verwenden kann



3rd Party Libraries, eigene Klassen

Container



Beziehen der Services auf  
denen wir aufbauen



Meine Klasse benötigt  
verschiedene andere Klassen  
damit sie korrekt funktioniert



- Was ist hier das Problem?

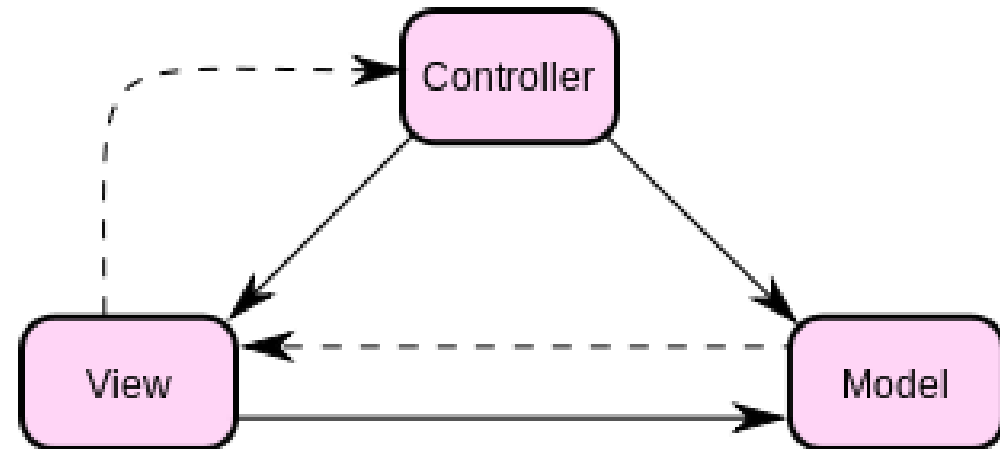
```
export class Car {  
  
  public engine: Engine;  
  public tires: Tires;  
  public description = 'No DI';  
  
  constructor() {  
    this.engine = new Engine();  
    this.tires = new Tires();  
  }  
  
  // Method using the engine and tires  
  drive() {  
    return `${this.description} car with ` +  
      `${this.engine.cylinders} cylinders and ${this.tires.make} tires.`  
  }  
}
```



- Schaut euch den Beispielcode zu Dependency Injection an
  - Analysiert was in inject.ts und injector.ts geschieht
  - Wie funktioniert der Example Code? Wie werden die Dependencies aufgelöst?
- Erstellt eine weitere Dependency und fügt diese der Klasse Car hinzu



- MVC (Mode-View-Controller)
  - Model  
Data Model for the View
  - View  
Displays the data Model
  - Controller  
Handles User Input  
Handles Business Logic



# Engineering



```
function Product(id, description) {
  this.getId = function() {
    return id;
  };
  this.getDescription = function() {
    return description;
  };
}

function Cart(eventAggregator) {
  var items = [];

  this.addItem = function(item) {
    items.push(item);
  };
}

var products = [
  new Product(1, "Star Wars Lego Ship"),
  new Product(2, "Barbie Doll"),
  new Product(3, "Remote Control Airplane")],
cart = new Cart();
```

```
(function() {
  function addToCart() {
    var productId = $(this).attr('id');
    var product = $.grep(products, function(x) {
      return x.getId() == productId;
    })[0];
    cart.addItem(product);

    var newItem = $('<li></li>')
      .html(product.getDescription())
      .attr('id-cart', product.getId())
      .appendTo("#cart");
  }

  products.forEach(function(product) {
    var newItem = $('<li></li>')
      .html(product.getDescription())
      .attr('id', product.getId())
      .dblclick(addToCart)
      .appendTo("#products");
  });
})();
```

```
function Product(  
  this.getId =  
    return id  
  };  
  this.getDescription =  
    return description;  
  };  
}  
  
var products = [  
  new Product(1, "Star Wars Lego Ship"),  
  new Product(2, "Barbie Doll"),  
];  
  
products.forEach(function(product) {  
  var newItem = $('<li></li>')  
    .html(product.getDescription())  
    .attr('id', product.getId())  
    .dblclick(addToCart)  
    .appendTo("#products");  
});  
});  
  
(function() {  
  function addToCart() {  
    var productId = $(this).attr('id');  
    var product = $.grep(products, function(x) {  
      return x.getId() == productId;  
    })[0];  
    cart.addItem(product);  
  
    var newItem = $('<li></li>')  
      .html(product.getDescription())  
      .attr('id-cart', product.getId())  
      .appendTo("#cart");  
  }  
})()
```

add the item to the cart  
when an item is clicked

updating the DOM to display  
the newly added item

populate the initial list of  
products on the page

MVC!

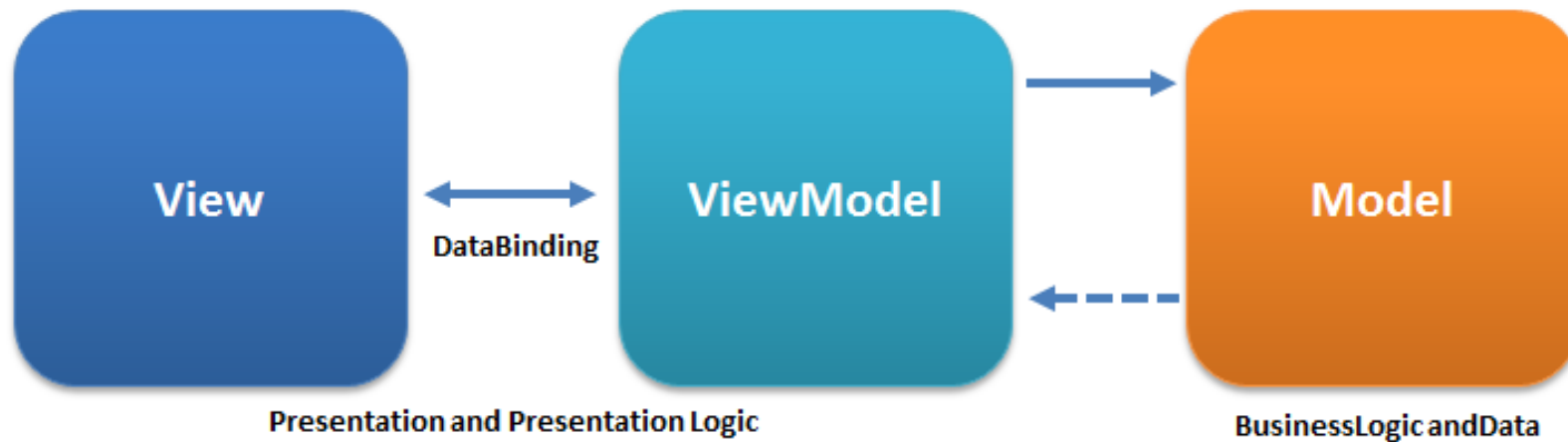
```
function Cart() {
    var items = [];

    this.addItem = function(item) {
        items.push(item);
        eventAggregator.publish("itemAdded", item);
    };
}

var cartView = (function() {
    eventAggregator.subscribe("itemAdded", function(eventArgs) {
        var newItem = $('<li></li>')
            .html(eventArgs.getDescription())
            .attr('id-cart', eventArgs.getId())
            .appendTo("#cart");
    });
})();

var cartController = (function(cart) {
    eventAggregator.subscribe("productSelected", function(eventArgs) {
        cart.addItem(eventArgs.product);
    });
})(new Cart());
```

- MVVM (Model-View-ViewModel)



- Dependency Management
  - Management of external libraries
  - Management of internal libraries





# Typescript

---

- Für Angular wird primär die Version 2+ von Typescript eingesetzt, welche Vereinfachungen für Typisierungen bringt
- Dem Compiler können nun Typen Informationen für externe Bibliotheken mitgegeben werden
- Installieren kann man die Typisierungen über npm
  - npm install @types/lib → @types/moment

```
"typeRoots": [  
  "../node_modules/@types"  
]
```

```
"types": [  
  "hammerjs",  
  "jasmine",  
  "node",  
  "protractor",  
  "selenium-webdriver",  
  "source-map",  
  "uglify-js",  
  "webpack"  
]
```

<https://www.typescriptlang.org/docs/handbook/typings-for-npm-packages.html>

- Die Information zu den Types wird in den compilerOptions des tsconfig.json angegeben
- Es gibt dabei die Möglichkeit die Typen explizit anzugeben oder auf einen Folder mit den Typeninformationen zu verweisen

```
"types": [  
  "hammerjs",  
  "jasmine",  
  "node",  
  "protractor",  
  "selenium-webdriver",  
  "source-map",  
  "uglify-js",  
  "webpack"  
]  
  
"typeRoots": [  
  "../node_modules/@types"  
]
```

# Links

---

- <http://www.typescriptlang.org/>
- <https://basarat.gitbooks.io/typescript/>